UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1302

Programme 2
Structures Nouvelles d'Ordinateurs

TECHNIQUES DE COMPILATION ET D'OPTIMISATION DE LA MEMOIRE ET DES REGISTRES SUR LE CRAY-2

Christine EISENBEIS
William JALBY
Alain LICHNEWSKY

Octobre 1990

RR-1382

TECHNIQUES DE COMPILATION ET D'OPTIMISATION DE LA MÉMOIRE ET DES REGISTRES SUR LE CRAY-2

COMPILER TECHNIQUES FOR OPTIMIZING MEMORY AND REGISTER USAGE ON THE CRAY-2

Christine EISENBEIS
William JALBY
Alain LICHNEWSKY

Résumé

Dans une étude précédente, nous avons montré l'efficacité de la méthode d'ordonnancement cyclique pour générer du code vectoriel performant pour l'architecture du CRAY-2, comparée aux compilateurs standards actuels. Cette méthode utilisait le cadre de la compaction de microcode, grâce à une modélisation adéquate du flot des instructions vectorielles du CRAY-2.

Dans ce papier, nous étudiöns plus précisément deux points: la modélisation de l'architecture de la machine et son impact sur la méthode d'ordonnancement cyclique envisagée, et le comportement de la technique spécifique d'allocation cyclique des registres, qui doit être particulièrement pointue pour tirer le meilleur parti du petit nombre de registres vectoriels. Nous présentons des comparaisons de nos méthodes avec celles utilisées dans le domaine des processeurs RISC et VLIW, ainsi que les performances du code obtenu.

Abstract

In a previous work [3], a cyclic scheduling method was shown efficient to generate vector code for the Cray-2 architecture, and compared to existing compilers. This method was using the framework of microcode compaction through a simplified model of the Cray-2 vector instruction stream. In this paper, we further elaborate on two issues: how to model the machine architecture within the underlying cyclic scheduling method, and the performance of the register allocation technique that must endeavour to make a good use of the scarce resource represented by vector registers. Comparisons with other related work in the area of RISC and VLIW processors are presented as well as performance data.

1 Introduction

The CRAY-2 architecture provides both vector registers and a local memory which must be used to hide the main memory latency and reduce the demand on main memory throughput to obtain very efficient programs. Similar characteristics occur for several modern vector processors, but this is much more critical on the CRAY-2 because it uses dynamic memory and a very short CPU cycle, as opposed to other processors like the Cray X-MP or Y-MP. To exploit this memory hierarchy, the code generator faces a significant resource allocation and scheduling problem, which is made even more critical by the absence of chaining in the CRAY-2 architecture.

Our approach to these problems relies on several ideas:

- the approximation of the NP-complete original optimization problem [5] [9] by a sequential procedure involving three natural subgoals: the functional unit scheduling, data spilling, and register allocation;
- a careful scheduling of the functional units performing vector operations in parallel;
- a register allocation procedure that attempts to make efficient use of the limited size of the register set;
- usage of a simplified model of the architecture, that permits concentration on the key architectural features and reduces the complexity of the involved algorithm.

This overall approach is described in our previous paper [3], and we will study here in more detail two topics whose importance has been confirmed by the experience we have gained. The first is the choice of the model used to present an abstract and simplified description of the machine. The second concerns the register allocation scheme we are using and a comparison of its characteristics with several alternatives, some of which have appeared in the literature.

Although our presentation is restricted to the optimization of vector loops for the CRAY-2 architecture, our techniques have much wider applicability and are of interest for RISC and VLIW architectures. The fact that the common memory latency of the CRAY-2 is much larger than on most current vector machines permits us to explore a direction of growing importance. Indeed, the relative importance of both memory delays and transmission latency is increasing because of the evolution of semiconductor technology and the decrease in cycle time. The architectural evolution toward larger multiprocessor configurations will also result in the usage of interconnection networks that will have longer latencies, and possibly very complex dynamic behaviour. Some of these characteristics are already present on the CRAY-2, and our experimental results show how they can be tackled.

Our presentation will adopt the following outline. In the first section, a model for the execution of the vector instructions on the CRAY-2 is introduced. It captures the RISC-like nature of the CRAY-2 vector architecture and allows use of a framework and techniques developed for code optimization on horizontally microcoded processors. Then in section 2 we present the principles of operation of our code optimizer (VASCO). In section 3, we analyze the impact of the simplifying assumptions of our model on the performance and we

show how this model can take into account parameters describing the common memory and its practical behavior. In section 4, our nonstandard register allocation strategy for loops is introduced and compared with previously published schemes. Among its advantages are the facts that it can be integrated as a post pass after functional unit scheduling and data spilling and that it requires a number of physical registers equal to the number of simultaneously alive virtual registers implied by the functional unit schedule [1]. Although the experimental results and the code optimizer used are specific to the CRAY-2, most of the ideas may be carried to other machines with similar memory hierarchies.

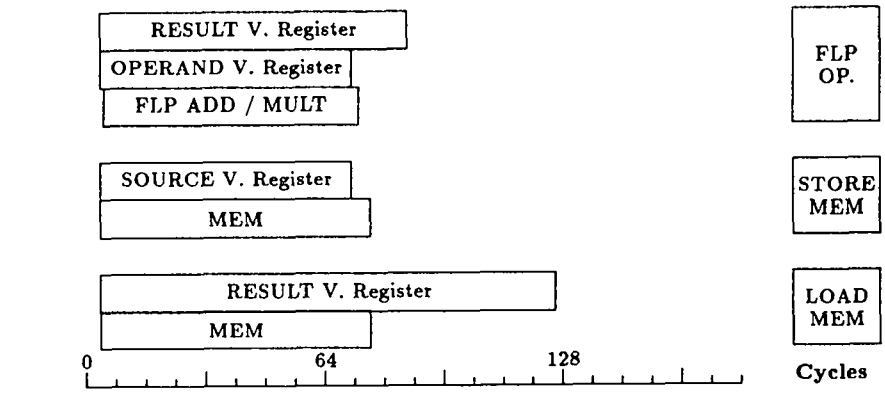
2 A model of the CRAY-2 architecture

In this section, we present a model of the instruction execution of the CRAY-2. The key point is to develop a model that has the ability to handle simultaneous execution of several instructions provided that they do not use the same hardware resource. Such a model is provided within the classical framework of microcode optimization, and enables us to use the techniques developed in that context [14] [5] [11] [9].

For the convenience of the reader, the main characteristics of the CRAY-2 are summarized hereafter. The main memory is shared by the four CPU's, organized in four quadrants linked to the processors by a X-bar switch, and involves adequate access arbitration and buffering mechanisms. Each CPU privately owns a local memory of 16 Kwords, eight vector registers of 64 elements each, eight floating point registers, and eight address registers. The transfers between all these storage elements are entirely managed by explicit transfer instructions emitted by the the compilers. The peak data rate from common memory is one word per cycle per processor, with a high latency (depending on the exact memory options). However, two independent vector memory accesses can be partially overlapped resulting in an "ideal" rate of 72 cycles per block of 64 elements (210 Mwords per second). The local memory provides the same peak access rate but has a much smaller latency , and the absence of conflicts guarantees a practical bandwidth equal to its peak. However, the aggregate data rate from these two memory levels is not sufficient to saturate the two floating point units, making use of the vector registers a crucial part of code optimization for the CRAY-2. The vector register file can support up to eight accesses simultaneously. Moreover, due to the lack of instruction chaining, the vector registers have to be managed very carefully as buffers in order to exploit the parallelism between the functional units.

A classical notion in the framework of microcode optimization is the reservation table which allows description of the use of resources for a series of discrete time steps. These tables specify, for each time step, the occupancy of the hardware resources subject to reservation. In addition, all the instructions (or microoperations) are associated with elementary reservation tables (called in the sequel templates) which describe the impact of the instruction execution on the reservation tables. Each template may specify reservations of several resources spanning several time steps. Scheduling the basic operations and checking for the

[1] In the absence of spilling, given a functional unit scheduling, it thus generates the minimal number of registers needed.



Cycle 0 = Instruction Issue

Figure 1: Instructions and Resource Reservations

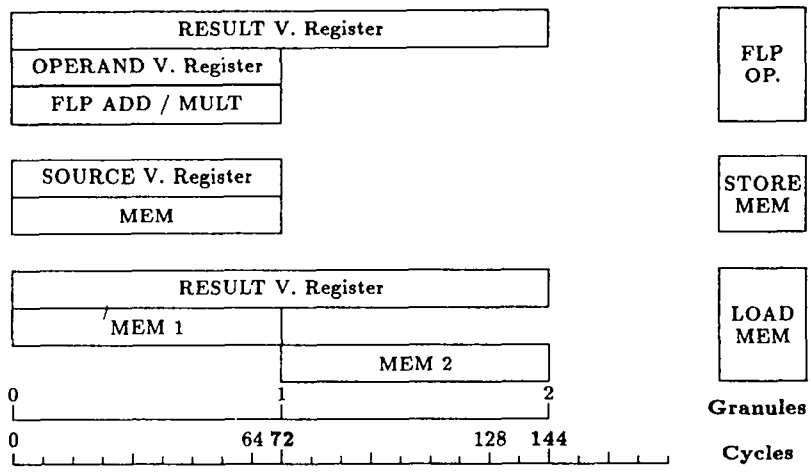
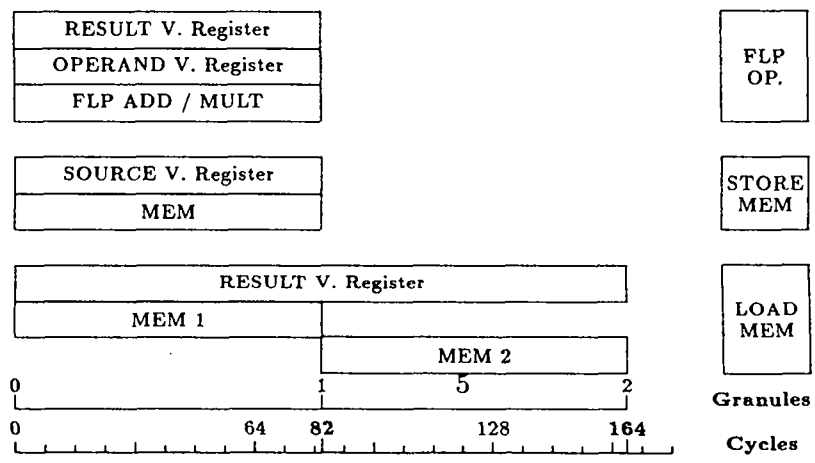


Figure 2: Templates with a macrocycle of 72 cycles



effect of interlocks now amounts to placing the templates onto the reservation table in a conflict-free manner. The role of the code optimizer is to determine the template scheduling without violating dependence and resource constraints while minimizing the execution time.

The key advantage of the microcode compaction framework is its use of reservation tables that can take precisely into account simultaneous activities of the various components of the machine. A first solution to embed the CRAY-2 architecture in such a model is to consider all the instructions (scalar and vector) and to associate with each of them a template describing it at the processor cycle level. Figure 1 shows such templates for the most common CRAY-2 vector instructions. Such a fine grain model was successfully used by Arya [1] for optimizing code for the Cray-1. The main drawback of such a solution is the cost of the resulting optimization phase: this phase amounts to solving optimization problems over the integer field which very quickly exhibit a high computational complexity.

We chose to go a step further in the modeling process by simplifying it to reduce the complexity of the optimization phase. We are also able to take advantage of this modeling strategy to describe the practical characteristics of main memory access. The rationale and key model characteristics are summarized hereafter.

- First, the model takes into account only vector operations with full vector length. As a result, we proceed in two phases. The first phase only considers vector instructions, schedules them, and allocates vector registers and vector temporaries. In a final pass, we emit and schedule scalar code. This is justified by the fact that we are primarily interested in vector loop optimization for which most of the time is spent in the vector instructions. Neglecting to optimize vector length less than 64 is not considered too severe since it occurs only once per loop after the classical strip mining loop transformation.

- Second, for the scheduling of the vector instructions, we first define a time unit called a macrocycle. All the timings during the scheduling of the vector instructions are expressed as integer multiples of this time unit. In Figure 1, we give the exact timings of the vector instructions together with their representations as templates using a macrocycle of 82 cycles (Cf. Figure 2) and a macrocycle of 72 cycles (Cf. Figure 1). The introduction of the macrocycle greatly simplifies the reservation tables, and therefore their optimization. However, for the scheduling of scalar instructions, we work with a time granularity of a single cycle, which is almost two orders of magnitude smaller. It should be noted that for instructions that do not involve common memory, the timings can easily be obtained from the manufacturer's specifications. The case of instructions involving common memory transactions is more complicated: due to conflicts, the real time spent in a vector load can vary largely. This implies that mapping loads or stores into templates requires approximations within the modeling procedure.

- Third, for scheduling, we will assume that the vector instruction issue is exactly described by the program's code according to our breakdown into macrocycles, and therefore entirely under our control. Several vector instructions can be executed at the same macrocycle provided there is no resource conflict. This differs from the actual hardware

issue policy, which is greedy, and therefore will start an instruction as soon as it will not cause any resource conflict.

For example, according to our model, we may suppose that at macrocycle i we issue a single instruction, A, and at macrocycle $i+1$ we issue the next vector instruction, B. The actual execution of such a sequence of instructions will fall into one of two cases: (1) there is a resource conflict which will prevent execution of instruction B at cycle i, and the instruction issue will exactly comply with our model, or (2) the opportunity of executing both instructions A and B within the same macrocycle i will cause a discrepancy between the modeled schedule and the actual execution. In the latter case, this greedy behavior will not add conflicts for following instructions. The semantic consistency of our models derives from the fact that there is only one memory port, and that the hardware uses register reservation to guarantee the correctness of the execution. These two features guarantee that all the memory instructions will be executed in a serial manner exactly in the same order as they have been issued [2].

The templates are derived directly from the description of the instructions by marking the resources occupied during each time macrocycle. The busy times are simply rounded up to the next granule. For example, Figure 2 (respectively Figure 1) shows the templates with a macrocycle of 82 (respectively 72) clock periods. The number 82 corresponds to the longest reservation time, in processor cycles, which occurs for the result register in a floating point add. (Cf. Figure 1). The number 72 corresponds to the minimal latency between successive vector memory access.

Due to the choice of the macrocycle (72 or 82), all the resources in the templates are considered as single-stage pipeline, except the common memory access, which is considered as a 2-stage pipeline. Thus, in the case of a main memory store or load, the access is split into two macrocycles and main memory accesses might be partially overlapped. The local memory access is single stage. We clearly see the advantage of the macrocycle that results in extremely simple templates (busy times of either one or two macrocycles), greatly simplifying the complexity of the vector scheduling phase at the price of an approximation of the processor timings.

For the vector scheduling phase (which only deals with macrocycles), the only difference between a macrocycle of 72 and 82 is the occupancy of the result register: two macrocycles in the first case, and one in the second case. However, this minor difference at the level of the resulting models hides deeper effects which will appear both at compile-time (register usage) and at run-time.

3 VASCO: a code optimizer for the CRAY-2

This section outlines the organization and the principles of VASCO, our code optimizer for the CRAY-2. The main focus will be on the scheduling problem; register management will be

[2] In the presence of parallel memory transfers (as in the Cray-XMP architecture) special instructions would have to be issued to keep the run-time pattern coherent with the data dependencies of the code.

```
      DO 1 i=1,1200,1
         z(i) = (a+x(i)*y(i))*b
1     CONTINUE
```

Figure 4: Fortran code of MV1

BODY

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
MEM1	**X1**	**Y1**		X2	Y2		X3	Y3		X4	Y4	**Z1**			Z2			Z3			Z4
MEM2		**X1**	**Y1**		X2	Y2		X3	Y3		X4	Y4									
ADD						+1			+2			+3			+4						
MUL				*1			*2	*1		*3	*2		*4	*3			*4				

Figure 5: Reservation table for MV1 (VAS72)

BODY

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
MEM1	**X1**	**Y1**		X2	Y2		X3	Y3	**Z1**			Z2			Z3	
MEM2		**X1**	**Y1**		X2	Y2		X3	Y3							
ADD					+1			+2			+3					
MUL				*1		*1	*2		*2	*3		*3				

Figure 6: Reservation table for MV1 (VAS82)

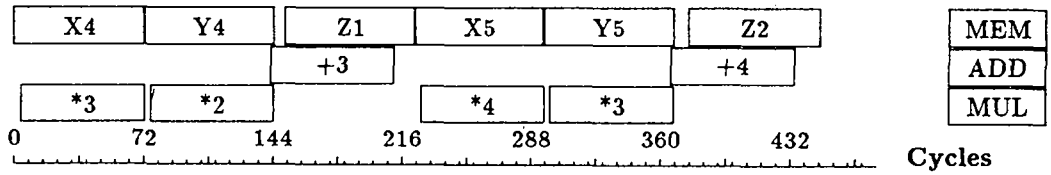


Figure 7: Timing execution of MV1 (VAS72)

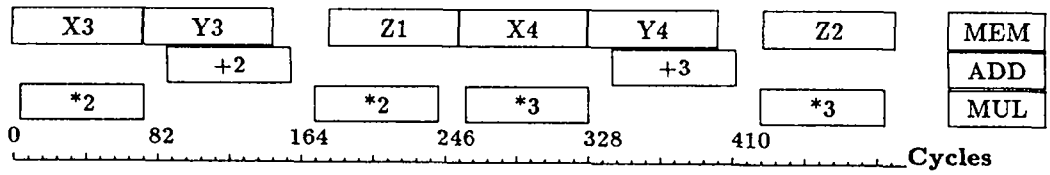


Figure 8: Timing execution of MV1 (VAS82)

developed in section 5. As explained in the previous section, our code optimization strategy distinguishes between vector operations and scalar operations: in a first coarse granularity pass, vector operations are scheduled, then, in a subsequent pass, scalar operations are scheduled, working at the machine cycle granularity. The reader should by now be aware of the approximate nature of our "optimization" procedure, and should acquire a deeper insight into the rationale of our approach by considering the related set of heuristics that supports the problem decomposition steps exposed below.

The optimization of vector operations itself is decomposed in two consecutive phases:

- scheduling of the functional units assuming an infinite number of pseudo-registers
- management of the storage units: spilling in local memory, physical register allocation.

During the first phase, the activities of the virtual registers are very precisely recorded, first to preserve the dependencies within the program, and second to gather all the information necessary for the allocation and spilling phases. The allocation of physical registers is performed after the functional unit scheduling for two major reasons: first, it avoids the premature introduction of unnecessary dependencies due to register reservations, and second, it simplifies the scheduling pass because we do not have to take into account reservation conflicts of the registers.

3.1 Preprocessing

We are working on a vector loop which has already been detected and processed by a vectorizer (in our case VATIL, [10]). The dependence graph [8] is available at this stage and several optimizations, including strip mining, have been performed. The dependence information (intra- and inter-iterations) is used during the scheduling process to preserve the semantics of the program. The strip mining operation consists of breaking (blocking) all the vectorized loops by blocks of 64 iterations. For a loop of N iterations, the result is an outermost loop of $\left\lceil \frac{N}{64} \right\rceil$ and an innermost loop of 64 iterations.

Because vector operations are considered atomic blocks, the term iteration (respectively loop) will exclusively denote the outermost iteration (respectively outermost loop).

3.2 Code generation of the loop body

In this phase, we generate an intermediate vector code for the loop body, which is used as a basic pattern for the cyclic scheduling of the iterations.

The registers are handled in a virtual manner according to a single assignment rule: the number of registers is assumed to be unbounded, and each time a register is needed to hold a value resulting from a load or an operation, a new register is allocated. In the cyclic scheduling, each iteration of the loop body will be represented by a similar copy of the generated intermediate code. The virtual registers defined in the generic loop body are named R_1 through R_k.The copy corresponding to iteration i will create the registers numbered $R_1(i)$ through $R_k(i)$. The virtual registers which either live on loop entry or are

invariant are named $R_l(0)$. Globally, as a consequence of the single assignment rule, each virtual register is written once, but may be read several times. The main advantage of such a technique is to avoid introduction of artificial dependencies due to bad allocation [2].

3.3 Scheduling

The scheduling of the whole loop involves two intimately related subproblems: scheduling of the generic loop body code and then scheduling of the successive iterations. We use a cyclic scheduling technique that was developed for array processor microcode compaction [11] [9] and tackles the two issues simultaneously.

In this method, all the iterations are exactly scheduled following the same pattern derived from the generic loop body. Therefore, each iteration contributes to the reservation table by a translated copy of the generic loop body reservation table RT. Iterations are started with a constant unknown period of d macrocycles. The global scheduling problem is to build a reservation table, RT, for the loop body and find a period, d, such that:

- the sequence $RT + kd$ does not conflict(at each macrocycle, a given functional unit is not used more than once).
- the semantic dependence constraints are satisfied.
- the total execution time is minimized.

Note that the execution of one iteration may span more than d macrocycles and therefore, at a given cycle, several successive iterations might be concurrently executing. Consequently, the local scheduling has to take *cyclic constraints* into account to avoid resource conflicts between iterations. This simply means that dependencies are satisfied under the periodic initiation mode, and cyclic constraints can readily be computed from the dependence graph as a function of d. The payoff of that additional complexity is the perfect chaining between iterations.

For determining d, we first compute for each functional unit, t, the number of macrocycles, n_t, where t is used during one iteration. The initial value for d is chosen to be the maximum value of n_t. This corresponds to the saturation of one of the functional units. Then for this period, d, we try to schedule the loop body according to a list scheduling strategy modified to take into account both semantic and cyclic constraints. If it appears that no cyclic scheduling is possible with period d, we increment it by one and try again. However, if the loop is vectorizable, it is always possible to find a cyclic scheduling with the initial value of d (for more details see [11], [9]). The technique used is the exact equivalent of the introduction of delays in a pipeline [13].

It should be noted that because we are dealing with virtual registers, the computation of d does not take into account the occupation of the registers. Moreover, the compaction process as described above is architecture independent in the sense that all the architectural characteristics (which are just used as parameters by the compaction procedure) are embedded into the template description. So, for example, our two different models of the CRAY-2,

which differ by the choice of the value of the macrocycle, will use the same scheduling procedure but with a different set of templates: VAS72 (respectively VAS82) corresponds to the version of the code optimizer using the templates as described in Figure 2 (respectively in Figure 3). An example of the result of the compaction procedure using the two models is given in Figures 5 and 6.

The success of such a strategy of scheduling is mainly due to templates' being extremely simple: most functional units are not used more than one macrocycle. Figures 9 and 10 give examples of the performance of the codes generated by VAS82 compared with codes obtained through the CFT77 3.0 compiler. Our code generation results in a better usage of the memory bandwidth. More systematic benchmarking results are given in [3].

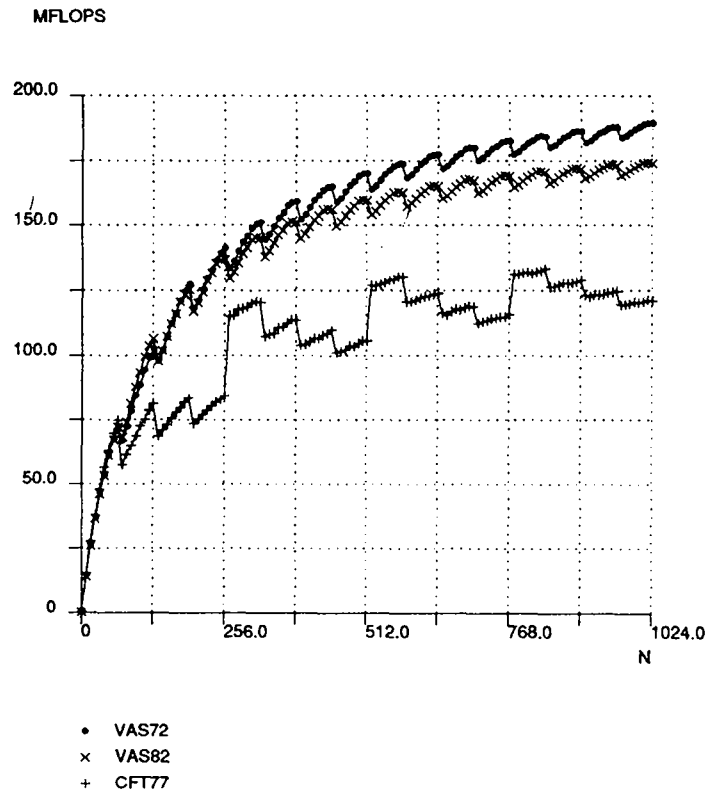


Figure 9: Performance of VAS72 and VAS82 compared with CFT77: MV1

3.4 Global register usage

During this step, we consider the schedule, (**$\mathcal{FU}$-sched**), obtained in the previous step as given, and generate all the information relative to register usage. This information will be used to find a register allocation scheme (cf. Section 5, which details register allocation) that allows execution as planned (i.e., according to (**$\mathcal{FU}$-sched**)). Clearly, it may become necessary to insert delays in the schedule or transfer data to and from local memory because

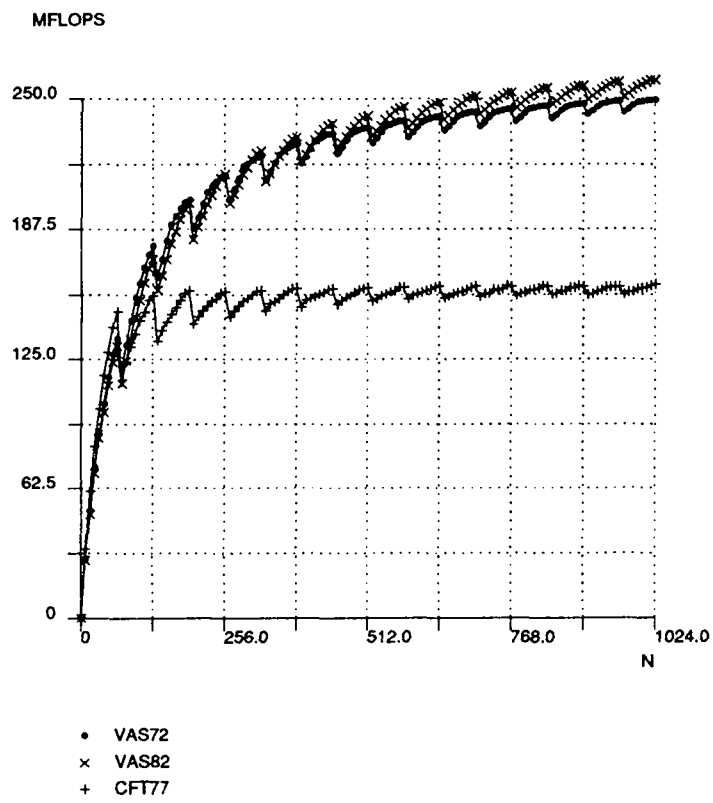


Figure 10: Performance of VAS72 and VAS82 compared with CFT77: DROT

of a shortage in registers, thus obtaining a schedule ($\mathcal{FU}$-**sched***). We are, in effect, decoupling the scheduling problem by imposing the constraint that ($\mathcal{FU}$-**sched***) is derived from ($\mathcal{FU}$-**sched**) by delay insertion and register spilling.

The birth date of a virtual register is defined as the cycle where it is assigned a value. Correspondingly, its death date is the cycle where its content is used for the last time. Between its birth and death, the virtual register is said to be live. At each cycle, the total number of live registers is determined, and the maximum number of live registers over the whole execution is called the critical register quantity (CRQ). Note that the number of live registers during a period of d cycles does vary during loop starting times and finishing times (cf Figure 12), and that CRQ corresponds to the steady state of the execution. This number clearly corresponds to the minimum[3] number of physical registers needed to execute the loop without spilling.

Many useful properties of registers are common to a register class and correspond to the virtual registers of the generic loop body. Specifically, the class R_k contains $R_k = \bigcup_{i=1..n} R_k(i)$.

In the scheduling procedure described above, there is no direct attempt to minimize

[3]Within our constrained framework relevant to ($\mathcal{FU}$-**sched***).

that quantity. However, as we observed in practice, the resulting CRQ is generally quite satisfactory. It should be noted that the quantity to be minimized is CRQ, not the lifespan of registers. Therefore, although they reduce the lifespan, techniques such as scheduling operations in reverse order do not necessarily reduce CRQ. To refine our approach, we are currently testing a level scheduling strategy which, instead of using an a priori ordering of the templates, allows us to opt for execution between several templates, according to some dynamically computed criteria. For example, the criteria used for selection might be the local minimization of the number of simultaneously alive registers.

4 Discussion of the model

The quality of the code produced by VASCO not only depends upon the algorithms and heuristics used for the optimization, but also upon our model of the CRAY-2 architecture. In this section we will analyze how the choice of the model influences the code generated and its performance. This last point raises the issue of evaluating how much the model differs from the real behavior.

M1	y(i)	=	(a * x(i)) + b
MV1	z(i)	=	((x(i) * y(i)) + a) * b
MVF1	t(i)	=	((x(i) * a) + y(i)) * z(i)
MCOM	zr(i)	=	xr(i) * yr(i) - xi(i) * yi(i)
	zi(i)	=	xr(i) * yi(i) + xi(i) * yr(i)
DROT	x(i)	=	c * z(i) - s * t(i)
	y(i)	=	s * z(i) + c * t(i)
MVC1	a(i)	=	a(i) + b(i,1) * c1
MVC2	a(i)	=	a(i) + b(i,1) * c1 + b(i,2) * c2
MVC3	a(i)	=	a(i) + b(i,1) * c1 + b(i,2) * c2 + b(i,3) * c3
MVC4	a(i)	=	a(i) + b(i,1) * c1 + b(i,2) * c2 + b(i,3) * c3 + b(i,4) * c4
MVC5	a(i)	=	a(i) + b(i,1) * c1 + b(i,2) * c2 + b(i,3) * c3 + b(i,4) * c4 + b(i,5) * c5
MVC6	a(i)	=	a(i) + b(i,1) * c1 + b(i,2) * c2 + b(i,3) * c3 + b(i,4) * c4 + b(i,5) * c5 + b(i,6) * c6
MVC7	a(i)	=	a(i) + b(i,1) * c1 + b(i,2) * c2 + b(i,3) * c3 + b(i,4) * c4 + b(i,5) * c5 + b(i,6) * c6 + b(i,7) * c7
MVC8	a(i)	=	a(i) + b(i,1) * c1 + b(i,2) * c2 + b(i,3) * c3 + b(i,4) * c4 + b(i,5) * c5 + b(i,6) * c6 + b(i,7) * c7 + b(i,8) * c8
MVC9	a(i)	=	a(i) + b(i,1) * c1 + b(i,2) * c2 + b(i,3) * c3 + b(i,4) * c4 + b(i,5) * c5 + b(i,6) * c6 + b(i,7) * c7 + b(i,8) * c8 + b(i,9) * c9
MVC10	a(i)	=	a(i) + b(i,1) * c1 + b(i,2) * c2 + b(i,3) * c3 + b(i,4) * c4 + b(i,5) * c5 + b(i,6) * c6 + b(i,7) * c7 + b(i,8) * c8 + b(i,9) * c9 + b(i,10) * c10
LL1	Lawrence Livermore Kernel 1		
LL7	Lawrence Livermore Kernel 7		
LL12	Lawrence Livermore Kernel 12		
LL21	Lawrence Livermore Kernel 21		
LL183	Third loop of Lawrence Livermore Kernel 18		

Table 1: Program kernels used in the experiments

Four major simplifications can be found in our model and need to be validated.

1. We used a model that is explicitly synchronized on macro-cycles intervals, although the execution appears asynchronous when analyzed at this level of granularity. This hides a more complex issue mechanism that works at a much lower time scale and involves a much more detailed description of the machine. In reality, we do not have absolute control over the execution of the instructions issued as soon as there is no resource conflict.

2. The loop optimization procedure takes into account only vector instructions; the scalar and address instructions are scheduled in a subsequent pass.

3. The choice of the macrocycle duration introduces some errors in that the timing of every instruction is rounded up to the next multiple of a macrocycle length (discretization error).

4. While the timing of non-memory instructions (instructions all of whose operands are in registers or local memory) can accurately be described, the timing of the common memory instructions has to be modeled: it is impossible to statically predict all memory conflicts, in addition to its being too complex.

In this section the impact on performance of each point mentioned above is analyzed in more detail. The experimental results were obtained using the codes described in Table 1. The sequence of kernels MVC1 through MVC10 was chosen because these kernels correspond to a gradual increase in the complexity of the loop body and allow a precise determination of the models' impact on resource usage.

4.1 Synchronous model

The choice of a "synchronous model" has a relatively minor impact on performance: the difference between our model and the real Cray issue mechanism is that some instructions will be issued earlier than assumed in our model. This greedy behavior will not result in generating conflicts later in the execution of the code. This is not true in general, as some timing anomalies may appear when instructions are issued earlier than planned [4]. However, such a situation does not occur for the CRAY-2 vector instructions due to the specific shape of their templates [4].

As a result, the total execution time observed in reality will be smaller than the one predicted, i.e., the code will perform better than expected. In practice, such a phenomenon occurs very rarely because most of the codes are memory bound on the CRAY-2. The presence of a single memory access unit implies that for vector loops, the code generated by our model will often contain a vector memory load or store every macrocycle. In such cases, the presence of these memory transactions in every macrocycle will limit the possible shift to one macrocycle and will only change the timing of the startup phase, not the steady state for the execution of the strip-mined loop.

Table 2: Latencies (in number of cycles) of two different models and impact of scalar instructions (stride 1); the presence of dag (resp. a double dag) following a code name indicating that VAS72 (resp. VAS82) required spilling.

Codes[4]	VAS72		VAS82	
	ScOv	*Chime*	*ScOv*	*Chime*
M1	0%	72	0%	74.5
MV1	2.5%	74.7	0.8%	81.6
MVF1	0.7%	73.25	0%	72
MCOM	0%	73.3	0.8%	81.3
DROT †	0.5%	71.2	1.2%	84.25
MVC1	0%	72	0.9%	73.6
MVC2	0%	72	0.6%	79
MVC3	0.4%	72.8	1%	82
MVC4 †	0.7%	62.8	1.2%	82.5
MVC5 †	1.2%	59.1	0.7%	81
MVC6 †	6.6%	71.4	1.8%	83
MVC7 †	4.9%	69.9	0.8%	82.3
MVC8 †	6.9%	64.2	0.2%	82.3
MVC9 †	9.4%	66.9	0.7%	82.3
MVC10 †	10.7%	70.2	0.5%	81.4
LL1 †	1.3%	74.7	0%	82.5
LL7 †‡	7.8%	73.9	1.3%	78.8
LL12	0.5%	72.8	0.5%	72.8
LL21	0%	72	0.7%	78.7
LL183	0%	72	0.4%	75.3

As an example, let us consider the timing diagrams obtained by simulation of the execution of MV1 codes (Figure 7 for VAS72 and Figure 8 for VAS82). In these diagrams, the row labeled MEM (resp. ADD and MUL) specifies the activity of the memory (respectively, adder and multiplier) unit. These diagrams were generated using the SIM facility, which takes into account most of the architectural features of the CRAY-2 except the memory conflicts between independent memory requests. It appears that the simulated behavior is very close to that produced by the models (cf. Figures 5 and 6); most of the differences are due to the discrepancy between the exact timings and the ones used in the model.

4.2 Impact of scalar instructions

Ignoring the scalar instructions in the optimization of the loop body greatly simplified the optimization procedure. This was justified by the fact that most scalar instructions take a number of cycles an order of magnitude smaller than the full vector length vector instructions. However, a legitimate question arises as to how much the post pass scheduling of the scalar instructions costs, and whether it excessively perturbates the execution of vector instructions. To obtain an experimental estimation of the cost, we first measured the latency (denoted $Lat1$ and expressed in cycles) between two consecutive blocks of 64 iterations for the codes generated by VASCO; these codes were then trimmed down by suppressing all

scalar instructions but the jumps and vector length decrementations (i.e., this corresponds to discarding almost all the scalar instructions), and the latency for these codes was measured (denoted $Lat2$). By comparing these two latencies and computing the scalar overhead $ScOv = (Lat1 - Lat2)/Lat1$, we obtained an estimate of the impact of the scalar instructions on the performance (cf. Table 2). It turns out that the cost in most cases is less than 5%, the only exceptions worth mentioning being codes involving spilling (these codes are marked with a single or a double dag in Table 2) for which the cost might be as high as 11 %. The reason for this is that, in these cases, the pointers to the local memory locations used for spill have to be spilled themselves due to the lack of address registers, which drastically increases the number of scalar instructions.

Table 3: Register requirements and memory bandwidth usage (stride 1, vector length = 1024 and timings performed in dedicated mode); the presence of dag (resp. a double dag) following a code name indicating that VAS72 (resp. VAS82) required spilling.

Codes	*CRQ*		*MWORDS*			*MFLOPS*		
	VAS72	*VAS82*	*VAS72*	*VAS82*	*CFT77*	*VAS72*	*VAS82*	*CFT77*
M1	6	4	185	176	148	185	176	148
MV1	6	5	188	174	118	188	174	118
MVF1	7	5	196	203	165	147	152	124
MCOM	8	8	193	177	143	193	177	143
DROT †	9	8	163	170	108	244	255	162
MVC1	6	4	201	197	143	100	98	71
MVC2	8	5	199	184	135	133	123	90
MVC3	8	6	197	180	143	148	135	107
MVC4 †	10	7	153	180	143	122	144	114
MVC5 †	10	8	154	180	143	128	150	120
MVC6 †	11	8	102	171	148	87	147	127
MVC7 †	11	8	115	172	151	101	150	132
MVC8 †	12	8	92	174	154	82	155	137
MVC9 †	12	8	97	174	156	87	157	140
MVC10 †	13	8	76	175	160	69	159	145
LL1 †	9	6	176	170	133	220	212	166
LL7 †‡	10	9	128	170	137	205	272	219
LL12	4	4	198	198	167	66	66	56
LL21	6	4	188	177	135	99	88	67
LL183	7	6	201	197	189	134	131	126

4.3 Choice of the macrocycle

The choice of both macrocycles (72 and 82 cycles) was first motivated by the resulting simplification of the templates, and therefore, the table reservation optimization process. Consequently, all the elementary reservation tables for both models were extremely simple. Although the difference between the two models may seem minor, the impact on the generated code and performance is far from being negligible. In the sequel, we shall designate by VAS72 (respectively VAS82) the result of applying VASCO with the 72 (respectively 82)

cycles macrocycle templates. CFT77 will stand for the output of the Cray CFT77 Version 3.0 compiler.

First, for the code optimizer, the only difference between the two models is that VAS72 considers the result register of the vector floating point add or multiply reserved for two macrocycles while VAS82 considers it reserved just for one macrocycle. As we expected, VAS72 turns out to be consuming more registers (see Table 3, the columns *CRQ* of which show the number of virtual registers simultaneously alive). As a result, codes produced by VAS72 will more often require spilling than codes produced by VAS82.

The situation with performance results is more subtle. Table 3 gives the memory bandwidths obtained when running codes obtained by VAS72, VAS82, and CFT77. The memory bandwidth was computed by dividing the total number of memory accesses actually performed by the time measured. The timings were done in a dedicated environment and on a CRAY-2 with a dynamic memory. Because all the codes in this table are memory bound, the Megaflop rate can be obtained by straightforward scaling. Depending upon the code, we distinguish two cases: either VAS72 was able to produce a code requiring no spilling (i.e., $CRQ \leq 8$) or not. In the first case, VAS72 performs better than VAS82, while in the second the situation is reversed. The second case is relatively easy to explain: the cost of the spilling code required by VAS72 reduces the performance so severely that the advantage of VAS72 is lost.

The situation in the first case (no spilling for VAS72) is directly related to the choice of the macrocycle and the induced approximation. For VAS72, the reservation time for both memory and functional unit is modeled accurately. The only major error introduced is in the reservation of the result registers which are considered two macrocycles (i.e., 144 cycles) although they are, in reality, reserved for only 82 cycles. As a result, the memory and the functional units are well modeled and their usage very well optimized via VAS72.

On the other hand, the choice of 82 as a macrocycle implies that memory and functional unit timing will be overestimated, resulting in an under-utilization of these resources. In contrast, the register reservations are more accurately modeled.

However, the impact of such an overestimation is not uniform, as evidenced by consideration of the following two extreme cases:

1. none of the instructions scheduled at macrocycle $i+1$ use a value produced by one of the floating point units at macrocycle i.
2. one of the instructions scheduled at macrocycle $i+1$ uses a result produced by a floating point unit at macrocycle i.

In the first case, although the model assumes a length of 82 cycles, all the instructions will be executed in around 72 cycles. In the second case, the presence of dependence and reservations involving the result register will enforce the value of 82 as the time spent at execution for macrocycle i. An example of such a case is depicted in Figure 8, where the addition (+2) of the second macrocycle depends on the multiplication (+2) executed during the previous cycle. Although the second memory access, Y3, is executed immediately following the first one (resulting in the full utilization of memory bandwidth), the third

memory access is delayed. This stems from the critical path of the code produced by VAS82 being constituted by the concatenation of triples of dependent floating point operations (*2, +2, *2). Conversely, VAS72 has scheduled the operations corresponding to the same triple in nonconsecutive macrocycles, resulting in a better utilization of the memory bandwidth (cf Figure 7).

This effect can be better appreciated if we compute the apparent chime, which is defined as the number of cycles between the starting of two consecutive blocks of 64 iterations of the loop body divided by the latency expressed in macrocycles, as predicted by the model (cf. Table 2).

For VAS72, when no spilling is required, the model is very close to the apparent chime because the critical path of code produced by VAS72 is mostly comprised of operations, all of which last 72 cycles. As described elsewhere [3], the spilling procedure we used introduces macrocycles during which no operations are scheduled: they correspond to waiting for the liberation of result registers. In such cases, the model is considerably inaccurate because such macrocycles are accounted for a full macrocycle, while in reality they last only about 10 cycles (82 -72).

For VAS82, the apparent chime rate varies between 72 and 82 cycles (cf Table 2). This variation is due to the phenomenon described above, i.e, the presence of dependencies between instructions scheduled in consecutive macrocycles. An extreme case is MVF1 where the apparent chime rate is 72, which turns out to be a result of the scheduling process that systematically inserted a macrocycle between the production of a result by one of the floating point units and its usage. On the other hand, for the loop body of MV1, any result produced by a functional unit was used in the following macrocycle, resulting in an apparent chime rate of 81.6 cycles.

4.4 Modeling the memory behavior

The two models we used were constructed assuming an almost ideal memory behavior in terms of bandwidth and assuming the processing of a vector memory request every 72 or 82 cycles. Although code optimized for such models may seem inadequate because of over optimistic memory behavior modeling, in practice it turned out to generate high performance code. The reason is that most of the codes are memory bound on the CRAY-2. Therefore, it is important to be able to issue memory transactions as soon as possible to saturate the memory bandwidth. Our models fulfill that requirement by trying to schedule a memory transaction every macrocycle, achieving a perfect overlap between the memory operations and the floating point computations. This also means that the constraints imposed by the model are more severe than in reality. If we assume that all vector memory accesses are more than 82 cycles long, it is easy to show that the critical path in the codes generated by VASCO will mostly be constituted by the memory access. To validate that assumption, we ran the generated codes with all the vector accesses performed with stride 2 (this is equivalent to a slowdown of the memory access by a factor of 3). As shown in Table 4, the codes produced by VAS72 and VAS82 are still saturating the available memory bandwidth.

Table 4: Memory bandwidth usage (stride 2, vector length 1024 and timings performed in dedicated mode)

Codes	*MWORDS*		
	VAS72	*VAS82*	*CFT77*
M1	66.8	68.7	59.5
MV1	67.3	68.5	55
MVF1	69.6	70.0	65.0
MCOM	68.1	68.1	59.4
DROT	68.9	68.7	55.3
MVC1	69.8	70.0	58.4
MVC2	68.9	69.7	58.7
MVC3	69.1	69.3	60.1
MVC4	67.9	68.7	59.7
MVC5	67.8	68.8	60.1
MVC6	63.0	68.2	61.1
MVC7	65.5	67.6	62.3
MVC8	60.7	67.8	63.2
MVC9	63.4	68.1	63.7
MVC10	54.2	67.8	64.2
LL1	66.6	68.2	63.6
LL7	63.3	68.4	59.3
LL12	69.6	69.6	64.5
LL21	67.3	67.7	57.9
LL183	68.7	69.4	69.4

4.5 Strict limitations of the common memory model

There are two issues of common memory access that are clearly overlooked by our simple model:

- non-unit strides memory accesses have been modeled, a priori, identically to unit-stride (or to even stride) accesses: the same load template is used for all cases. Although such a choice does not waste memory cycles as shown in the previous subsection for the stride 2 case (i.e., the code generated still saturates the memory bandwidth available), it may result in wasting register space. This could be avoided in many cases where stride is known at compile-time (either explicitly or after global interprocedural analysis) by using different templates for loads with strides resulting in a degradation of the memory performance (i.e., stride multiples of 2, 4, 8 etc...).
- Our pipelined model ignores many of the fine grain details of processor and memory system architecture: bank level reservation, data path buffering, hardware resources multiplexing (like pseudo-banking). This can be justified in light of the satisfactory use we can make of this model in our search for faster codes. However, we can also show examples where our generated code clearly shows that the limit of the simplified model has been reached.

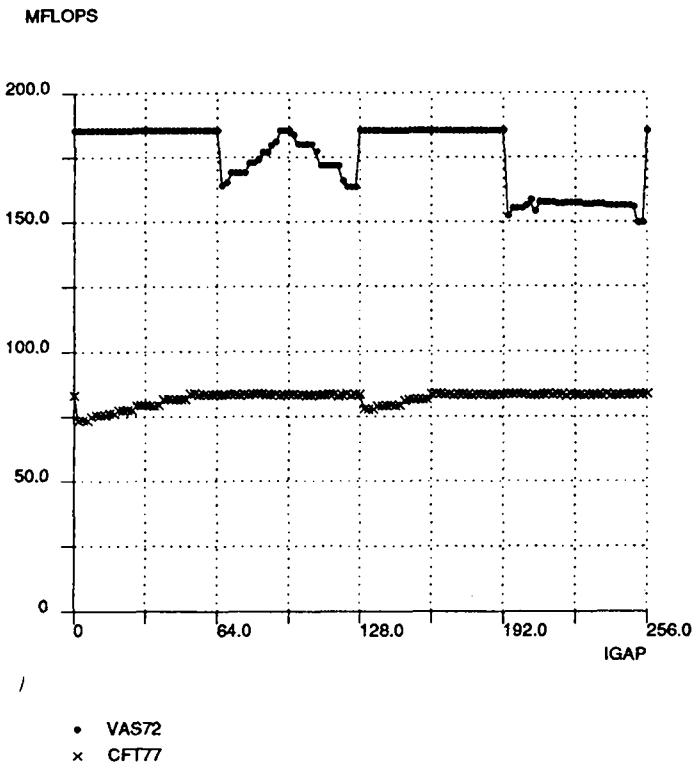


Figure 11: Influence of the gap on performance

It is interesting to note that a very simple experiment suffices to illustrate this point. It can be performed by using the following piece of code

```
c     0 < IGAP < 512
c     N < 4096  therefore no dependency
      DO I = 1, N
1         A(I+IGAP+4096) = (B + A(I)) * C
```

where IGAP is a parameter influencing the memory offset of memory writes with respect to memory reads[5].

The results (Figure 11), obtained in a dedicated mode with dynamic memory configuration, show that our optimized code is sensitive to changes in IGAP, whereas CFT77 code does not stress memory access enough to exhibit the phenomenon.

Such an effect is relatively difficult to take into account at compile-time because in most cases, it depends upon the data layout. The only effective solution is to unroll the loop and try to schedule successive loads on the same vector into consecutive macrocycles, although the price in terms of register space is prohibitive when compared with the benefits.

[5]This also offsets quadrant controllers and pseudo banking common hardware (modulo 4 and modulo 256).

5 Optimization of register usage

During the scheduling of the functional units, we were dealing with virtual registers to focus on optimizing the usage of the functional units. Now physical registers need to be allocated. Even if the number of virtual registers simultaneously alive is less than the number of physical registers, the task is complex because the loop structure and the cyclic scheduling technique used induces cyclic constraints on the physical register assignment.

As an example, let us assume a loop for which the functional unit schedule has a latency $L = 3$ and for which the reservation of the virtual register is given in Figure 12. The solution of allocating, for every iteration i, $R_1(i)$ to V0 would not work: starting the second iteration three cycles later than the first one will lead to a conflict between $R_1(1)$ and $R_1(2)$ which are mapped on the same physical register V0. To avoid the reservation conflict, the second iteration needs to be started more than three cycles later, in fact seven cycles later due to the reservation time of $R_2(1)$. As a result, the schedule computed previously is no longer valid because it was assuming a latency of three cycles. Computing a new schedule with a latency of seven would not fix the problem because the reservation of the virtual registers might change and furthermore it would result in an unacceptable slowdown. More generally, such a problem occurs each time the lifetime of a virtual register exceeds the latency between two consecutive iterations of the loop.

The solution proposed by Lam [9] consists of allocating to virtual registers within a same class, but associated with different iterations, different physical registers, which is achieved by first unrolling the object code of the loop and performing the allocation for that unrolled loop. Unrolling the loop k times multiplies the latency by k while keeping the same reservation timings for the virtual registers. The underlying principle of this method can also be described by stating that instead of looking for an allocation mapping ϕ identical for each iteration, we are looking for a mapping ϕ and a period u such that:

For every virtual register family R_k *and for every iteration* i

$$\phi(R_k(i+u)) = \phi(R_k(i))$$

In fact, the quantity u corresponds to the unrolling factor.

In the sequel three different allocation strategies are presented and compared. For comparison, we will use two main criteria: the number of physical register used and the unrolling factor. The first criterion is clearly crucial because registers are a scarce resource on the CRAY-2. In practice, the only point that really matters is that the number of physical registers used is less than or equal to the number of physical registers available. The second criterion is related to the size of the code generated: unrolling the loop k times multiplies the size of the code by k which on some machines with a limited instruction cache or buffer may have a adverse effect on performance. The influence of exceeding the aggregate size of the four CRAY-2 instruction buffers results in a performance degradation in the order of 5%, as indicated by our practical experience.

For a unified presentation of the three algorithms, we will define a common framework. Let us suppose that the latency between two iterations is L. We recall that virtual registers

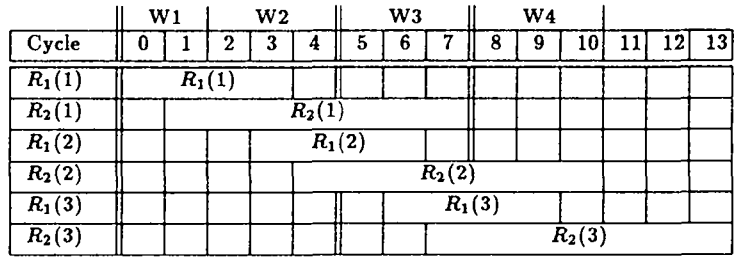


Figure 12: Virtual Register Reservation

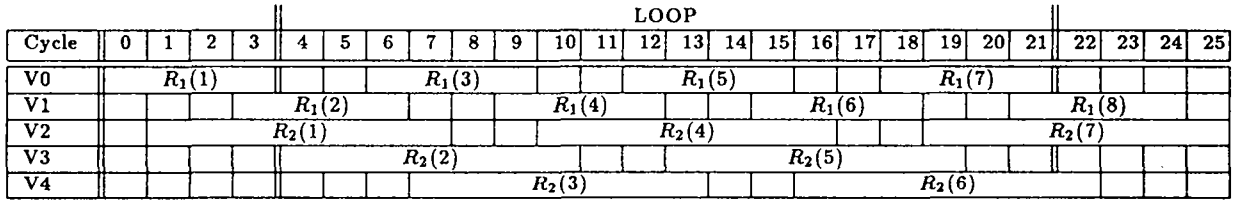


Figure 13: Register allocation by Algorithm 1

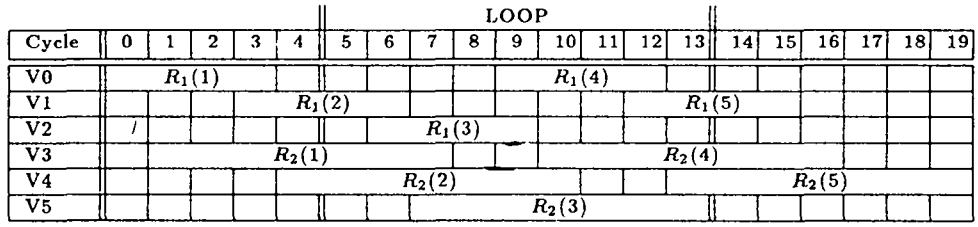


Figure 14: Register allocation by Algorithm2

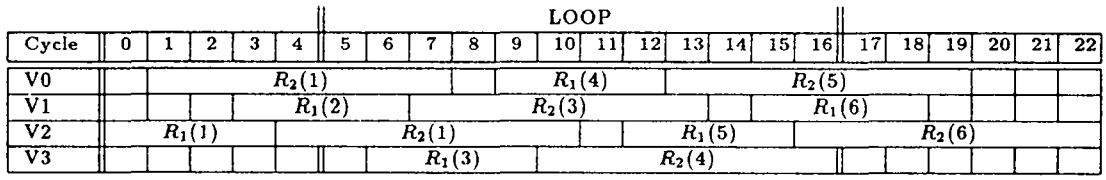


Figure 15: Register allocation by Algorithm 3

are grouped into families $(R_1 \cdots R_l)$ and that iteration i uses virtual registers $R_1(i) \cdots R_l(i)$. With each virtual register $R_k(i)$, we will associate its corresponding lifetime duration expressed in cycles. Because this quantity only depends upon the family, we will note it N_k and CRQ will denote the maximum number of virtual registers simultaneously alive ($CRQ = 4$ for the example described in Figure 12). We will assume that there are CRQ physical registers available numbered $0 \cdots CRQ - 1$. The function ϕ (called allocation function) will denote a mapping in $0 \cdots CRQ - 1$, which gives for each virtual register the physical register it is assigned to.

5.1 Algorithm 1

This method is a systematic application of the *Modulo variable expansion* as proposed by Lam [9]. The key point is that *if members of a register class R_k have a lifetime N_k and if an iteration is initiated every L cycles, then at least $\lceil N_k/L \rceil$ members of that class have to be kept alive concurrently in that many physical registers.* This register allocation problem can be solved by first unrolling the loop $s = \lceil N_k/L \rceil$ times and then dedicating s distinct physical registers for the allocation of the corresponding virtual registers $R_k(i) \cdots R_k(i+s)$. More precisely, the virtual register $R_k(i)$ is assigned to the (i modulo s)th register allocated to the class R_k. More generally, this has to be applied to every register family. Thus, the loop is unrolled u_1 times with $u_1 = lcm_k\ (\lceil N_k/L \rceil)$ and the total number of registers used will be $RU_1 = \sum_k \lceil N_k/L \rceil$. In the example (Cf. Figure 12) $N_1 = 4$ and $N_2 = 7$ implies that the loop has to be unrolled $u_1 = lcm(2,3) = 6$ times and that five registers are needed. The complete allocation is given in Figure 13.

5.2 Algorithm 2

The previous algorithm may result in a very large degree of unrolling, so Lam proposed a modification of the algorithm that reduces the degree of unrolling at the price of using, a priori, more registers. First the loop is unrolled u_2 times with $u_2 = max_k \lceil N_k/L \rceil$ and then to each virtual register class R_k, t physical registers are allocated where t is the smallest factor of u_2 that is no smaller than $\lceil N_k/L \rceil$, that is:

$$t = min_n\ n \text{ such that } n \geq \lceil N_k/L \rceil \text{ and } u_2 \equiv 0 \pmod{n} \qquad (1)$$

To reduce the number of registers used a post pass was added to take care of registers whose lifetime was less than L; they were first sorted by decreasing lifetime and allocated to the first physical register available (cf [9]).

In the example (Cf. Figure 12), this gives an unrolling factor of 4, requires six physical registers, and the allocation scheme is described in Figure 14.

5.3 Algorithm 3

Our basic algorithm, which will use exactly CRQ registers, was introduced in [3] and is described below. First, consider the loop as entirely unrolled and decompose this infinite

sequence into a sequence of consecutive chunks with length L cycles called windows (W1, W2, W3, etc ... in the example). Let us note that the first may have less than L cycles, although all the subsequent will contain exactly L cycles. Except for the first windows, all the windows will exhibit exactly the same reservation tables, modulo a translation in time, thus describing a periodic steady state. One of them is then selected in the steady state region as a reference window (W2 in our example).

Our method consists of building a function ϕ, on a window by window basis, satisfying the following constraints.

1. Over each window considered separately, the function ϕ is an admissible register allocation: any pair of virtual registers simultaneously alive in any cycle of the window will be assigned via ϕ different physical registers.

2. Between two consecutive windows, the function ϕ preserves the continuity of the assignment: a virtual register whose life spans several windows is assigned across all the windows to the same physical register.

It can easily be checked that such a function ϕ will constitute an admissible assignment for the whole loop.

The procedure for building ϕ reflects exactly the two properties mentioned above. First we build a partial assignment ϕ_r over the reference window. This problem can be solved exactly in polynomial time using exactly CRQ registers. More precisely, the algorithm is linear with respect to the number of virtual registers alive inside the reference window. In our example, let us pick W3 as the reference window, then over W3, five registers, $R_2(1)$, $R_1(2)$, $R_2(2)$, $R_1(3)$ and $R_2(3)$, are alive (although not more than four are simultaneously alive at any cycle). For our example, the initial assignment over W3 was given by

$$\begin{aligned} \phi_r(R_2(1)) &= 0 \quad \phi_r(R_1(2)) = 1 \\ \phi_r(R_2(2)) &= 2 \quad \phi_r(R_1(3)) = 3 \\ \phi_r(R_2(3)) &= 1 \end{aligned}$$

For simplicity's sake, let us assume that the windows are numbered consecutively starting by the reference window (W1 is the reference window). The allocation over the other windows will be built according to the following rule:

$$\begin{aligned} &\textit{for any register name } R_k(u) \textit{ alive in any window } W_i \\ &\qquad \phi(R_k(u)) = \sigma^{i-1} \circ \phi_r(R_k(u-i+1)) \end{aligned}$$

where σ is a permutation over $0 \cdots CRQ-1$. It should be noted that due to the cyclic nature of our functional unit scheduling, $R_k(u)$ alive in W_i implies that $R_k(u-i+1)$ is alive during W_1. By such a construction the first property is automatically verified. As a consequence of such a construction, ϕ will necessarily be periodic with a period u_3 equal to the order of the permutation σ (i.e. $\sigma^{u_3} = Id$).

Now the problem is to build σ such that the continuity property holds. Let us first observe that over the reference window, two different names belonging to the same class

may appear, such as $R_1(2)$ and $R_1(3)$ in our example. These virtual registers need to be allocated to different physical registers.

To satisfy the continuity property, we need to look closely at the registers whose lives span over several windows. Let us introduce the set of virtual registers (denoted $RIGHT$), whose life spans over the reference window and the next one:

$$RIGHT = \{R_{i_1}(j_1), \cdots R_{i_q}(j_q)\} \tag{2}$$

Similarly, we define $LEFT$ as the set of virtual registers alive in the reference window and the previous one. Due to the cyclic nature of the loop and of the scheduling generated, there is a one to one mapping between the elements of $LEFT$ and $RIGHT$ associating names in the same class such that every element of $RIGHT$ can be extended by an element of $LEFT$. More precisely, if $R_t(u)$ appears in the set $RIGHT$, $R_t(u-1)$ is in $LEFT$ and is the "extension" of $R_t(u)$.

For each pair of elements associated in this manner $(R_{i_t}(k_t), R_{i_t}(k_t - 1))$, we consider the corresponding physical register numbers to which they were allocated over the reference window. We can define a partial mapping θ over $0 \cdots CRQ - 1$ by:

$$\theta(\phi_r(R_{i_t}(k_t))) = (\phi_r(R_{i_t}(k_t - 1))) \tag{3}$$

This partial mapping θ is injective and can be completed into a bijection σ. It can easily be checked that such a permutation will satisfy the continuity constraint.

In our example, the sets $RIGHT$ and $LEFT$ are given by

$$RIGHT = \{R_2(2), R_1(3), R_2(3)\}$$
$$LEFT = \{R_2(1), R_1(2), R_2(2)\}$$

The couples associated are $(R_2(2), R_2(1))$, $(R_1(3), R_1(2))$, $(R_2(3), R_2(2))$. Then θ is given by

$$\theta(\phi_r(R_2(2))) = \theta(2) = \phi_r(R_2(1)) = 0$$
$$\theta(\phi_r(R_1(3))) = \theta(3) = \phi_r(R_1(2)) = 1$$
$$\theta(\phi_r(R_2(3))) = \theta(1) = \phi_r(R_2(2)) = 2$$

There are many ways to complete θ in order to get σ. For this purpose, we build the partial graph of θ and complete it to limit the length of the cycles. The reason is that the order of σ, and therefore the unrolling factor u_3, is the least common multiple of the length cycles. By the same token, for the eight registers of the CRAY, the maximal u_3 is 15. To reduce this unrolling, we apply the previous algorithm not with CRQ as a target, but with the number of physical registers, which gives more room for determining ϕ_r, limiting the number of edges in the partial graph of θ and reducing a priori the unrolling factor. Another technique used for reducing u_3 is to choose the reference window such that it minimizes the number of elements of the set $RIGHT$, thereby possibly reducing the order of σ. As a corollary, this implies that the maximum degree of unrolling is bound by the maximal order

of a permutation over $CRQ - 1$ elements: for the CRAY2, this reduces the maximal degree of unrolling down to 12.

In our example, σ was chosen as

$$\sigma(2) = \theta(2) = 0 \quad \sigma(3) = \theta(3) = 1$$
$$\sigma(1) = \theta(1) = 2 \qquad \sigma(0) = 3$$

It should be noted than even if all the register lifetimes are less than the latency L, which will result in no unrolling for Algorithm 1 and 2, Algorithm 3 may generate unrolling because it tries systematically to minimize the number of registers used.

As a final point, one of the major advantages of this algorithm is that the number of registers it requires only depends on CRQ. Therefore, after the functional unit scheduling phase, two situations may arise: either CRQ is less than 8 so we can apply directly the allocation procedure as stated in Algorithm 3, or CRQ is greater than 8. In this latter case, we first perform a spilling pass in order to reduce CRQ down to 8, then we allocate the physical registers. The spilling strategy we used is relatively simple and is a simple variant of the one described in [3].

5.4 Comparison

Table 5: Comparison of different physical register allocations (VAS72)

Codes	*Lat*	*CRQ*	u_1	u_2	u_3	RU_1	RU_2	RU_3
M1	2	6	2	2	2	6	6	6
MV1	3	6	2	2	2	7	7	7
MVF1	4	7	2	2	2	8	8	8
MCOM	6	8	1	1	2	10	10	8
DROT †	5	8 (9)	2	2	2	9	9	8
MVC1	3	6	2	2	2	6	6	6
MVC2	4	8	2	2	2	8	8	8
MVC3	5	8	2	2	2	11	11	8
MVC4 †	9	8 (10)	1	1	2	14	10	8
MVC5 †	11	8 (10)	1	1	2	17	9	8
MVC6 †	16	8 (11)	1	1	3	23	10	8
MVC7 †	16	8 (11)	1	1	2	26	10	8
MVC8 †	25	8 (12)	1	1	2	32	10	8
MVC9 †	25	8 (12)	1	1	2	35	10	8
MVC10 †	34	8 (13)	1	1	3	41	9	8
LL1 †	4	8 (9)	2	2	6	10	10	8
LL7 †	16	8 (10)	1	1	6	26	11	8
LL12	3	4	2	2	2	4	4	4
LL21	3	6	2	2	2	6	6	6
LL183	6	7	2	2	2	10	9	8

Tables 5 and 6 give the unrolling degrees and the number of registers used for each of the three algorithms described above. The column labeled u_1 (resp. u_2 and u_3) indicates the

degree of unrolling resulting from using Algorithm 1 (resp. 2 and 3). Similarly the columns RU_1, RU_2, and RU_3 indicate the number of physical registers required. For the codes which a priori required spilling, we first performed the spilling pass before the allocation. In Tables 5 and 6, these codes are associated with two numbers in the column CRQ: the first one is the value of CRQ after the spilling pass, the second one (in parenthesis) is the original value of CRQ before spilling.

Algorithm 1 exhibits a moderate degree of unrolling but is clearly not competitive due to the prohibitive number of registers it requires. Algorithm 2 gives exactly the same degree of unrolling as Algorithm 1. This is because the life of all register names was not spreading over more than two windows (i.e., $N_k/L \leq 2$); for such a special case, the unrolling degree of both Algorithms 1 and 2 are identical because they result in exactly the same modulo variable expansion ($lcm(1,2) = max(1,2) = 2$). However, Algorithm 2 is consuming less registers, primarily due to the special postpass for the registers whose lifetime is less than the latency.

Finally, as expected, Algorithm 3 is making the best usage of the available registers. The price to be paid is not prohibitive: the unrolling degree of Algorithm 3 is in most cases very similar to the one produced by Algorithm 2 (cf. Tables 5 and 6).

Table 6: Comparison of different physical register allocations (VAS82)

Codes	*Lat*	*CRQ*	u_1	u_2	u_3	RU_1	RU_2	RU_3
M1	2	4	2	2	2	5	5	5
MV1	3	5	2	2	2	7	7	7
MVF1	4	5	1	1	1	6	6	6
MCOM	6	8	1	1	1	10	9	8
DROT	4	8	2	2	2	10	9	8
MVC1	3	4	1	1	1	4	4	4
MVC2	4	5	1	1	1	7	6	7
MVC3	5	6	1	1	1	10	7	8
MVC4	6	7	1	1	1	13	7	8
MVC5	7	8	1	1	1	16	8	8
MVC6	8	8	1	1	1	19	8	8
MVC7	9	8	1	1	1	22	8	8
MVC8	10	8	1	1	1	25	8	8
MVC9	11	8	1	1	1	28	9	8
MVC10	12	8	1	1	2	31	8	8
LL1	4	6	1	1	1	8	7	8
LL7 ‡	11	8 (9)	1	1	2	26	10	8
LL12	3	4	2	2	2	4	4	4
LL21	3	4	1	1	1	4	4	4
LL183	6	6	1	1	1	8	6	8

6 Conclusion

We have shown the applicability of a technique based on the microcompaction framework for generating efficient vector code on the CRAY-2. The experimental results presented demonstrate the need for a sophisticated register allocation procedure efficient enough to make good use of the vector registers which appear to be a very scarce resource on the CRAY architectures.

The modeling procedure we used gives some insight on the relation of key architectural features to practical performance of general vector code. This can be used to improve the tradeoff that has to be made between purely architectural considerations and compiler related issues, following a methodology much popularized by the RISC movement.

More generally, supercomputers are increasingly using complex hierarchical memory systems to achieve a data rate matching the arithmetic rate. Although such memory organizations result in a substantial improvement at least in peak performance, the efficiency may become highly dependent upon the usage of the different storage levels of the hierarchy. This implies a major change in the algorithm design, the larger applicability of high-level restructuring transformations, and an evolution of compiler technology. At that level, several problems have to be solved:

- code optimization when memory may have a very long, unpredictable and highly fluctuating latency;
- optimization of the usage of all the various levels of storage;
- minimization of the transfers between the different levels, which includes generation and scheduling of the corresponding code.

Acknowledgements

The authors thank Edward Davidson for his valuable comments about possible timing anomalies which may arise when instructions are issued earlier than planned. The authors would also like to thank Mrs Becky Gering for her editorial assistance.

References

[1] Arya, S., *Optimal Instruction Scheduling for a Class of Vector Processors: an Integer Programming Approach*, Report CRL-TR-19-83, University of Michigan, 1983

[2] Cytron, R., Ferrante, J., *What's in a name? The value of renaming for parallelism detection and storage allocation*, Proc. ICPP, 1987

[3] Eisenbeis, C., Jalby, W., Lichnewsky, A., *Squeezing more CPU performance out of a Cray-2 by vector block scheduling*, Proc. Supercomputing 88, Kissimee, Florida 1988.

[4] Eisenbeis, C., Jalby, W., Lichnewsky, A., *Compile-time optimization of memory usage on the CRAY2*, Proc. NATO Supercomputer Workshop, Trondheim, Norway 1989.

[5] Fisher, J.A., *The optimization of horizontal microcode within and beyond basic blocks: an application of processor scheduling with resources*, Phd thesis, New York Univ, 1979.

[6] Fisher, J.A., Ellis, J.R., Ruttenberg, J.C., Nicolau, A., *Parallel processing: a smart compiler and a dumb machine*, Proc. SIGPLAN Symp. on Compiler Construction, 1984.

[7] Kennedy, K., *Automatic Vectorization of Fortran Programs to Vector Form*, Technical Report, Rice University, Houston, TX, October 1980.

[8] Kuck, D.J., Kuhn, R., Padua, D., Leasure, B., Wolfe, M., *Dependence Graphs and Compiler Optimizations*, Proc. 8th ACM Symp. POPL, Williamsburgh, VA, 1981.

[9] Lam, M.S.L., *A systolic array optimizing compiler*, Phd Thesis, Carnegie Mellon University, 1987.

[10] Lichnewsky, A., Thomasset, F., *Techniques de base pour l'exploitation automatique du parallelisme dans les programmes*, Rapport de Recherche INRIA, N 460, 1985.

[11] Lichnewsky, A., Thomasset, F., Eisenbeis, C., *Automatic Detection of Parallelism in Scientific Programs with Application to Array-Processors*, Proc.of IBM Institute, North Holland, 1986.

[12] Nicolau, A., *Uniform Parallelism Exploitation in Ordinary Programs*, Proc. of ICPP, 1985.

[13] Patel, J.H., Davidson, E.S., *Improving the throughput of a pipeline by insertion of delays*, Proc., 3rd Ann. Symp. Comp. Arch., 1976.

[14] Touzeau, R.F., *A Fortran Compiler for the FPS-164 Scientific Computer*, SIGPLAN Notices, Vol. 19, N 6, 1984.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249 - 6399